IBM High Performance Computing Toolkit

I-Hsin Chung
IBM T.J. Watson Research Center
ihchung@us.ibm.com
What is it?

- **IBM long-term goal:**
  - An automatic performance tuning framework
    - Assist users to identify performance problems
    - Provide possible solutions
  - A common application performance analysis environment across all HPC platforms
  - Look at all aspects of performance (communication, memory, processor, I/O, etc) from within a single interface

- **Where we are: one consolidated package**
  - One consolidate package (AIX, Linux/Power)
  - Tools for MPI, OMP, processor, memory etc
  - Operate on the binary and yet provide reports in terms of source-level symbols
  - Dynamically activate/deactivate data collection and change what information to collect
  - One common visualization GUI
IBM High Performance Computing Toolkit on Blue Gene

- MPI performance: MPI Profiler/Tracer
- CPU performance: Xprofiler, HPM
- Threading performance: OpenMP profiling
- I/O performance: I/O profiling
- Visualization and analysis: PeekPerf
Supported Platforms

- AIX
- Linux
  - PowerPC
  - Blue Gene /L and Blue Gene /P
  - Intel x86 & AMD (planned)
- Eclipse integration
- Windows (Intel/AMD) + Mac (coming soon)
  - Offline Peekperf visualization capability only
AGENDA

- **Xprofiler**: call-graph profiling
- **HPM**: hardware counter data
- **MPI Profiler/Tracer**: MPI profiling
- **PompProf**: OpenMP profiling
- **MIO**: I/O profiling and optimization
- **IBM HPC Toolkit**
- **Questions/Comments**
XProfiler
Xprofiler

- Visualizer CPU time profiling data
- Compile and link with -g -pg flags + optimization
- Code execution generates gmon.out file
  - MPI applications generate gmon.out.1, ..., gmon.out.n
- Analyze gmon.out file with Xprofiler
  - xprofiler a.out gmon.out
- Important factors:
  - Sampling interval is in the order of ms
  - Profiling introduces overhead due to function calls
Xprofiler - Initial View

Clustered functions

Library calls
Xprofiler - Unclustering Functions

on “Filter” menu select “Uncluster Functions”
Xprofiler - Full View - Application and Library Calls

Program: stim0  Total CPU Usage: 1.31 seconds (summary of 1 gmon.out profile files)
Display Status: showing 95 out of 95 nodes and 96 out of 96 arcs
Xprofiler - Hide Lib Calls Menu

Now select “Hide All Library Calls”

Can also filter by: Function Names, CPU Time, Call Counts
Xprofiler - Application View

- Width of a bar: time including called routines
- Height of a bar: time excluding called routines
- Call arrows labeled with number of calls
- Overview window for easy navigation (View → Overview)
Xprofiler: Zoom In

Program: m6.profile  Total CPU Usage: 5832.88 seconds (summary of 1 gnup.out profile files)
Display Status: showing 25 out of 128 nodes and 50 out of 170 arcs
Xprofiler: Flat Profile

- **Menu Report** provides usual gprof reports plus some extra ones
  - Flat Profile
  - Call Graph Profile
  - Function Index
  - Function Call Summary
  - Library Statistics

<table>
<thead>
<tr>
<th>Cumulative</th>
<th>Self</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>%time</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>62.9</td>
<td>15.64</td>
<td></td>
</tr>
</tbody>
</table>
Xprofiler: Source Code Window

- Source code window displays source code with time profile (in ticks=0.01 sec)

- Access
  - Select function in main display
  - context menu
  - Select function in flat profile
  - Code Display
  - Show Source Code

```c
202 #define I2MNT(k,n) (i+1)*n+k
203 /\* use 2x-unrolling of the outer two loops */
204 /\* \*------------------------------------------------------------------*/
205 for (i=0; i<10; i++)
206 {'
207 for (j=0; j<10; j++)
208 {'
209    t11 = c[i+j];
210    t12 = c[i+j+1];
211    t21 = c[(i+1)*n+j];
212   }
213   t22 = c[(i+1)*n+(i+1)];
214   for (k=0; k<10; k++)
215   {'
216     t22 = t22 + a[(i+1)*n+k]*bt[j*n+k];
217     t21 = t21 + a[(i+1)*n+k]*bt[(i+1)*n+k];
218   }  
219   c[i+j] = t11;
220   c[i+j+1] = t12;
221   c[(i+1)*n+j] = t21;
222   c[(i+1)*n+(i+1)] = t22;
223   for (j=0; j<10; j++)
224   {'
225     t11 = c[i+j];
226     t21 = c[(i+1)*n+j];
227   for (k=0; k<10; k++)
228   {'
229     t11 = t11 + a[i+1]*nbt[j*n+k];
230     t21 = t21 + a[(i+1)*n+k]*bt[j*n+k];
231     c[i+j] = t11;
232     c[(i+1)*n+j] = t21;
233   }  
234   c[i+j] = t11;
235   c[(i+1)*n+j] = t21;
236   }
```
# Xprofiler - Disassembler Code

## Disassembler Code for .calc8

<table>
<thead>
<tr>
<th>Address</th>
<th>Address</th>
<th>Instruction</th>
<th>Assembler Code</th>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10002E18</td>
<td>81</td>
<td>FCA28C</td>
<td>fnms</td>
<td>POLD(I,J) = P(I,J) + ALPHA*(PNEW(I,J) -</td>
</tr>
<tr>
<td>10002E1C</td>
<td>64</td>
<td>C900008</td>
<td>lfd</td>
<td>UOLD(I,J) = U(I,J) + ALPHA*(UNEW(I,J) -</td>
</tr>
<tr>
<td>10002E20</td>
<td>187</td>
<td>C900008</td>
<td>1fd</td>
<td>V(I,J) = VNEW(I,J)</td>
</tr>
<tr>
<td>10002E24</td>
<td>53</td>
<td>C9750008</td>
<td>1fd</td>
<td>VOLD(I,J) = V(I,J) + ALPHA*(VNEW(I,J) -</td>
</tr>
<tr>
<td>1002E28</td>
<td>89</td>
<td>FD6382A</td>
<td>fa</td>
<td>POLD(I,J) = P(I,J) + ALPHA*(PNEW(I,J) -</td>
</tr>
<tr>
<td>10002E2C</td>
<td>63</td>
<td>FD28387C</td>
<td>fnms</td>
<td>UOLD(I,J) = U(I,J) + ALPHA*(UNEW(I,J) -</td>
</tr>
<tr>
<td>10002E30</td>
<td>4</td>
<td>D00000000</td>
<td>stfd</td>
<td>VOLD(I,J) = V(I,J) + ALPHA*(VNEW(I,J) -</td>
</tr>
<tr>
<td>10002E34</td>
<td>4</td>
<td>C9540008</td>
<td>1fd</td>
<td>P(I,J) = PNEW(I,J)</td>
</tr>
<tr>
<td>10002E38</td>
<td>113</td>
<td>FCA302A</td>
<td>fa</td>
<td>UOLD(I,J) = U(I,J) + ALPHA*(UNEW(I,J) -</td>
</tr>
<tr>
<td>10002E3C</td>
<td>27</td>
<td>C0760008</td>
<td>1fd</td>
<td>VOLD(I,J) = V(I,J) + ALPHA*(VNEW(I,J) -</td>
</tr>
<tr>
<td>10002E40</td>
<td>87</td>
<td>FD8012FA</td>
<td>fma</td>
<td>POLD(I,J) = P(I,J) + ALPHA*(PNEW(I,J) -</td>
</tr>
<tr>
<td>10002E44</td>
<td>35</td>
<td>D0B00008</td>
<td>stfd</td>
<td>UOLD(I,J) = U(I,J) + ALPHA*(UNEW(I,J) -</td>
</tr>
<tr>
<td>10002E48</td>
<td>4</td>
<td>FC6382A</td>
<td>fa</td>
<td>VOLD(I,J) = V(I,J) + ALPHA*(VNEW(I,J) -</td>
</tr>
<tr>
<td>10002E4C</td>
<td>12</td>
<td>CD8A0008</td>
<td>1fd</td>
<td>P(I,J) = PNEW(I,J)</td>
</tr>
<tr>
<td>10002E50</td>
<td>62</td>
<td>F00218A</td>
<td>fma</td>
<td>UOLD(I,J) = U(I,J) + ALPHA*(UNEW(I,J) -</td>
</tr>
<tr>
<td>10002E54</td>
<td>36</td>
<td>C8500008</td>
<td>1fd</td>
<td>VOLD(I,J) = V(I,J) + ALPHA*(VNEW(I,J) -</td>
</tr>
<tr>
<td>10002E58</td>
<td>244</td>
<td>DC6C0008</td>
<td>stfd</td>
<td>P(I,J) = PNEW(I,J)</td>
</tr>
<tr>
<td>10002E5C</td>
<td>28</td>
<td>FD0040FA</td>
<td>fma</td>
<td>UOLD(I,J) = U(I,J) + ALPHA*(UNEW(I,J) -</td>
</tr>
<tr>
<td>10002E60</td>
<td>4</td>
<td>C0900008</td>
<td>1fd</td>
<td>VOLD(I,J) = V(I,J) + ALPHA*(VNEW(I,J) -</td>
</tr>
<tr>
<td>10002E64</td>
<td>316</td>
<td>DC040008</td>
<td>stfd</td>
<td>P(I,J) = PNEW(I,J)</td>
</tr>
<tr>
<td>10002E68</td>
<td>29</td>
<td>FC62507C</td>
<td>fnms</td>
<td>UOLD(I,J) = U(I,J) + ALPHA*(UNEW(I,J) -</td>
</tr>
</tbody>
</table>
Xprofiler: Tips and Hints

- monenable() / mondisabale() to sample certain regions
- Simplest when gmon.out.*, executable, and source code are in one directory
  - Select “Set File Search Path” on “File” menu to set source directory when source, and executable are not in the same directory
  - Can use -qfullpath to encode the path of the source files into the binary
- By default, call tree in main display is “clustered”
  - Menu Filter \rightarrow Uncluster Functions
  - Menu Filter \rightarrow Hide All Library Calls
- Libraries must match across systems!
  - on measurement nodes
  - on workstation used for display!
- Must sample realistic problem (sampling rate is 1/100 sec)
HPM – HW counter library
HPM: What Are Performance Counters

- Extra logic inserted in the processor to count specific events
  - Updated at every cycle
  - **Strengths:**
    - Non-intrusive
    - Very accurate
    - Low overhead
  - **Weakness**
    - Provides only hard counts
    - Specific for each processor
    - Access is not well documented
    - Lack of standard and documentation on what is counted
HPM: Hardware Counters

- 8 counters on PPC970 and Power4, 6 counters on Power 5/5+
  - Several (100+) events per counter
- 48 UPC counters on Blue Gene/L, 328 events
- 256 UPC counters on Blue Gene/P, ~1000 events

• Events can not be selected independently
  - PPC970: 41 groups, default: 23
  - Power 4: 64 groups, default: 40
  - Power 5: 140 groups (AIX 5.2), 148 groups (AIX 5.3), default 137
  - Power 5+: 152 groups, default 145
  - Blue Gene/L: 16 groups
  - Blue Gene/P: 4 groups
HPM: Hardware Counters

- Cycles
- Instructions
- Floating point instructions
- Integer instructions
- Load/stores
- Cache misses
- TLB misses
- Branch taken / not taken
- Branch mispredictions

- Useful derived metrics
  - IPC - instructions per cycle
  - Float point rate (Mflip/s)
  - Computation intensity
  - Instructions per load/store
  - Load/stores per cache miss
  - Cache hit rate
  - Loads per load miss
  - Stores per store miss
  - Loads per TLB miss
  - Branches mispredicted %

- Derived metrics allow users to correlate the behavior of the application to one or more of the hardware components

- One can define threshold values acceptable for metrics and take actions regarding program optimization when values are below the threshold
Derived Metrics

- Utilization rate
- Total FP load and store operations
- MIPS
- Instructions per cycle/run cycle/load store
- % Instructions dispatched that completed
- Fixed point operations per Cycle or load/stores
- Branches mispredicted percentage
- number of loads per load miss
- number of stores per store miss
- number of load/stores per L1 miss
- L1 cache hit rate
- number of loads per TLB miss
- number of loads/stores per TLB miss

- Total Loads from L2
- L2 load traffic
- L2 load bandwidth per processor
- Estimated latency from loads from L2
- % loads from L2 per cycle
- Total Loads from local L2
- local L2 load traffic
- local L2 load bandwidth per processor
- Estimated latency from loads from local L2
- % loads from local L2 per cycle
- Total Loads from L3
- L3 load traffic
- L3 load bandwidth per processor
- Estimated latency from loads from L3
- …
CPU/Memory Performance

- Instrumentation library
- Provides performance information for instrumented program sections
- Supports multiple instrumentation sections
- Supports MPI, threading and mixed mode
- Multiple sections may have the same ID
- Run-time performance information collection
LIBHPM

- Allows to go in the source code and instrument different sections independently
- Supports Fortran, C, and C++
- For each instrumented section provides:
  - Total count & duration (wall clock time)
  - Hardware performance counters information
  - Derived metrics
- Provides resource usage statistics for the total execution of the instrumented program
- Supports:
  - MPI, OpenMP, & pThreads
  - Multiple instrumentation points
  - Nested instrumentation
  - Multiple calls to an instrumented point
Event Sets

- 4 sets (0-3); ~1000 events
- Information for
  - Time
  - FPU
  - L3 memory
  - Processing Unit
  - Tree network
  - Torus network
Instrumentation section

hpmlInit(tasked, "my program");

hpmlStart(1, "outer call");
do_work();
hpmlStart(2, "computing meaning of life");
do_more_work();
hpmlStop(2);
hpmlStop(1);
hpmlTerminate(taskID);
Using LIBHPM

- **Declaration:**
- **Use:**
  - `#include f_hpm.h`
  - `call f_hpminit( 0, “prog” )`
  - `call f_hpmstart( 1, “work” )`
  - `do`
    - `call do_work()`
    - `call f_hpmstart( 22, “more work” )`
    - `call compute_meaning_of_life()`
    - `call f_hpmstop( 22 )`
  - `end do`
  - `call f_hpmstop( 1 )`
  - `call f_hpmterminate( 0 )`
OpenMP/Threading

- Thread-safe libhpm supports OpenMP and threaded applications.
- A thread-safe linker invocation, such as xlc_r and xlf_r, should be used or
-/libpthreads.a must be included in the list of libraries.
HPM: Multi-thread Support
Functions

- **hpmlInit( taskID, progName ) / f_hpminit( taskID, progName )**
  - taskID is an integer value indicating the node ID.
  - progName is a string with the program name.

- **hpmsStart( instID, label ) / f_hpmstart( instID, label )**
  - instID is the instrumented section ID. It should be > 0 and <= 100 (can be overridden)
  - Label is a string containing a label, which is displayed by PeekPerf.

- **hpmsStop( instID ) / f_hpmstop( instID )**
  - For each call to hpmsStart, there should be a corresponding call to hpmsStop with matching instID

- **hpmsTerminate( taskID ) / f_hpmtterminate( taskID )**
  - This function will generate the output. If the program exits without calling hpmsTerminate, no performance information will be generated.
Overhead

- libhpm collects information and performs summarization during run time
  - there can be considerable overhead if instrumentation sections are inserted inside inner loops.
- Guideline
  - If the overhead is several orders of magnitude smaller than the total duration of the instrumented section, you can safely ignore the overhead timing.
  - If the overhead is in the same order as the total duration of the instrumented section, you should be suspicious of the results.
  - If the overhead is within 20% of the measured wall clock time, a warning is printed to the ASCII output file.
C and C++ example

declaration:
#include "libhpm.h"

use:
hpmInit( tasked, "my program" );
hpmStart( 1, "outer call" );
do_work();
hpmStart( 2, "computing meaning of life" );
do_more_work();
hpmStop( 2 );
hpmStop( 1 );
hpmTerminate( taskID );
Fortran example

declaration:
#include "f_hpm.h"
use:
call f_hpminit( taskID, "my program" )
call f_hpmstart( 1, "Do Loop" )
do …
call do_work()
call f_hpmstart( 5, "computing meaning of life" );
call do_more_work();
call f_hpmstop( 5 );
end do
call f_hpmstop( 1 )
call f_hpmterminate( taskID )
Multithreaded program

!$OMP PARALLEL
!$OMP&PRIVATE (instID)
instID = 30+omp_get_thread_num()
call f_hpmstart( instID, "computing meaning of life" )
!$OMP DO
do ...
do_work()
end do
call f_hpmstop( instID )
!$OMP END PARALLEL

- If two threads use the same ID numbers for call to hpmTstart or hpmTstop, libhpm exits with the following error message:
  - hpmcount ERROR - Instance ID on wrong thread
Output

- **HPM_OUTPUT_NAME**
  - The name `<name>` is expanded into different file names:
    - `<name>.hpm` is the file name for ASCII output, which is a one-to-one copy of the screen output.
    - `<name>.viz` is the file name for XML output.

- **HPM_UNIQUE_FILE_NAME**
  - The following string is inserted before the last dot (.) in the file name:
    - `_<hostname>_<process_id>_<date>_<time>`
Considerations for MPI parallel programs

- **HPM\_AGGREGATE**
  - does aggregation
  - restricting the output to a subset of MPI tasks
  - takes a value, which is the name of a plug-in that defines the aggregation strategy

- **plug-in**
  - a shared object
  - *distributor* and *aggregator*
Distributor

- A subroutine that determines the MPI task ID
- Sets or resets environment variables accordingly
- Environment variable can be any environment variable
- The distributor is called before any environment variable is evaluated by HPM
Aggregator

- Aggregation of the hardware counter data across the MPI tasks
- After the hardware counter data is gathered
- Before the data is printed
- Before the derived metrics are computed
- Check Redbook for detailed interface description
Plug-in shipped with HPCT

- **mirror.so**
  - the plug-in that is called when no plug-in is requested.

- **loc merge.so**
  - does a local merge on each MPI task separately

- **single.so**
  - does the same as mirror.so, but only on MPI task 0. The output on all other tasks is discarded.

- **average.so**
  - a plug-in for taking averages across MPI tasks
Hardware Counter Performance Visualization
MPI Profiler/Tracer
Message-Passing Performance

MPI Profiler/Tracer

- Implements wrappers around MPI calls using the PMPI interface
  - start timer
  - call pmpi equivalent function
  - stop timer
- Captures MPI calls with source code traceback
- No changes to source code, but MUST compile with -g
- Microsecond order of magnitude overhead per MPI call
- Does not synchronize MPI calls
- Compile with -g and link with libmpitrace.a
- Generate XML files for peekperf
Message-Passing Performance

MPI Tracer

- Captures “timestamped” data for MPI calls with source traceback
- Provides a color-coded trace of execution
- Very useful to identify load-balancing issues
Compiling and Linking

- Consider turning off or having a lower level of optimization (-O2, -O1,...)
  - High level optimization affects the correctness of the debugging information and can also affect the call stack behavior.

- To link the application with the library
  - The option `-L/path/to/libraries`, where `/path/to/libraries` is the path where the libraries are located
  - The option `-lpitrace`, which should be before the MPI library `-lmich`, in the linking order
  - The option `-llicense` to link the license library
Environment Flags

- **TRACE_ALL_EVENTS** (default yes)
  - saves a record of all MPI events one after MPI Init(), until the application completes or until the trace buffer is full.
  - By default, for MPI ranks 0-255, or for all MPI ranks, if there are 256 or fewer processes in MPI_COMM_WORLD.
  - Alternative: trace_start/stop()

- **MAX_TRACE_EVENTS** (Default: 30,000)

- **TRACE_ALL_TASKS**
  - Set to “yes” to trace all tasks/ranks

- **MAX_TRACE_RANK**

- **TRACEBACK_LEVEL** (Default: 0)
  - Level of trace back the caller in the stack
  - Used to skipped wrappers

- **TRACE_SEND_PATTERN**
  - Has to set to “yes” to trace communication pattern (e.g., Average hops)
  - AverageHops = sum(Hopsi × Bytesi)/sum(Bytesi)
MPI Profiler Output

```
DATA VISUALIZATION WINDOW

<table>
<thead>
<tr>
<th>Label</th>
<th>Count</th>
<th>WallClock</th>
<th>Transferred Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUMMARY</td>
<td>1000</td>
<td>0.009871</td>
<td>2.21952e+07</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>1000</td>
<td>0.025688</td>
<td>2.21952e+07</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>1000</td>
<td>0.025688</td>
<td>2.21952e+07</td>
</tr>
<tr>
<td>neighbours(mhdf)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>50</td>
<td>0.000579</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>836</td>
<td>0.000894</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>659</td>
<td>0.000426</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>674</td>
<td>0.000443</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>658</td>
<td>0.001828</td>
<td>4.8152e+06</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>721</td>
<td>0.001852</td>
<td>4.8152e+06</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>771</td>
<td>0.002238</td>
<td>4.8152e+06</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>776</td>
<td>0.002336</td>
<td>4.8152e+06</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>823</td>
<td>0.00313</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>834</td>
<td>0.000426</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>867</td>
<td>0.000753</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>872</td>
<td>0.000592</td>
<td>614400</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>714</td>
<td>0.000746</td>
<td>4.8152e+06</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>719</td>
<td>0.002894</td>
<td>4.8152e+06</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>769</td>
<td>0.000673</td>
<td>4.8152e+06</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>774</td>
<td>0.002405</td>
<td>4.8152e+06</td>
</tr>
</tbody>
</table>

```

```
temp261(length2+1) = geq(i,jsta,7,2)
end do

CALL
MPI_RECV(temp1r2,length3,MPI_DOUBLE_PRECISION,jprev)

```

```
CALL
MPI_SEND(temp2s1,length3,MPI_DOUBLE_PRECISION,jprev)
```

```
CALL
MPI_SEND(temp2s1,length3,MPI_DOUBLE_PRECISION,jprev)
```

```
CALL
MPI_RECV(temp1r2,length3,MPI_DOUBLE_PRECISION,jprev)
```

```
CALL
MPI_RECV(temp1r2,length3,MPI_DOUBLE_PRECISION,jprev)
```
MPI Profile Visualization
MPI Trace Visualization
## MPI Message Size Distribution

<table>
<thead>
<tr>
<th>MPI Function</th>
<th>#Calls</th>
<th>Message Size</th>
<th>#Bytes</th>
<th>Walltime</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>1 (1)</td>
<td>0 ... 4</td>
<td>0</td>
<td>1E-07</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>1 (1)</td>
<td>0 ... 4</td>
<td>0</td>
<td>1E-07</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>2 (1)</td>
<td>0 ... 4</td>
<td>3</td>
<td>0.000006</td>
</tr>
<tr>
<td></td>
<td>2 (2)</td>
<td>5 ... 16</td>
<td>12</td>
<td>1.4E-06</td>
</tr>
<tr>
<td></td>
<td>2 (3)</td>
<td>17 ... 64</td>
<td>48</td>
<td>1.3E-06</td>
</tr>
<tr>
<td></td>
<td>2 (4)</td>
<td>65 ... 256</td>
<td>192</td>
<td>1.3E-06</td>
</tr>
<tr>
<td></td>
<td>2 (5)</td>
<td>257 ... 1K</td>
<td>768</td>
<td>1.3E-06</td>
</tr>
<tr>
<td></td>
<td>2 (6)</td>
<td>1K ... 4K</td>
<td>3072</td>
<td>1.3E-06</td>
</tr>
<tr>
<td></td>
<td>2 (7)</td>
<td>4K ... 16K</td>
<td>12288</td>
<td>1.3E-06</td>
</tr>
<tr>
<td></td>
<td>2 (8)</td>
<td>16K ... 64K</td>
<td>49152</td>
<td>1.3E-06</td>
</tr>
<tr>
<td></td>
<td>2 (9)</td>
<td>64K ... 256K</td>
<td>196608</td>
<td>1.7E-06</td>
</tr>
<tr>
<td></td>
<td>2 (A)</td>
<td>256K ... 1M</td>
<td>786432</td>
<td>1.7E-06</td>
</tr>
<tr>
<td></td>
<td>1 (B)</td>
<td>1M ... 4M</td>
<td>1048576</td>
<td>9E-07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI Function</th>
<th>#Calls</th>
<th>Message Size</th>
<th>#Bytes</th>
<th>Walltime</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Irecv</td>
<td>2 (1)</td>
<td>0 ... 4</td>
<td>3</td>
<td>4.7E-06</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (2)</td>
<td>5 ... 16</td>
<td>12</td>
<td>1.4E-06</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (3)</td>
<td>17 ... 64</td>
<td>48</td>
<td>1.5E-06</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (4)</td>
<td>65 ... 256</td>
<td>192</td>
<td>2.4E-06</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (5)</td>
<td>257 ... 1K</td>
<td>768</td>
<td>2.6E-06</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (6)</td>
<td>1K ... 4K</td>
<td>3072</td>
<td>3.4E-06</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (7)</td>
<td>4K ... 16K</td>
<td>12288</td>
<td>7.1E-06</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (8)</td>
<td>16K ... 64K</td>
<td>49152</td>
<td>2.23E-05</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (9)</td>
<td>64K ... 256K</td>
<td>196608</td>
<td>9.98E-05</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>2 (A)</td>
<td>256K ... 1M</td>
<td>786432</td>
<td>0.00039</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>1 (B)</td>
<td>1M ... 4M</td>
<td>1048576</td>
<td>0.000517</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>21 (1)</td>
<td>0 ... 4</td>
<td>0</td>
<td>1.98E-05</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>5 (1)</td>
<td>0 ... 4</td>
<td>0</td>
<td>7.8E-06</td>
</tr>
</tbody>
</table>
# Communication Summary

Communication summary for all tasks:

- Minimum communication time = 0.015 sec for task 0
- Median communication time = 5.016 sec for task 30
- Maximum communication time = 5.016 sec for task 20

<table>
<thead>
<tr>
<th>taskid</th>
<th>xcoord</th>
<th>ycoord</th>
<th>zcoord</th>
<th>procid</th>
<th>total_comm(sec)</th>
<th>avg_hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.015</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5.016</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5.016</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

\[
\text{AverageHops} = \frac{\sum_{i} Hops_i \times Bytes_i}{\sum_{i} Bytes_i}
\]
Configuration

- **Reduce data volume**
  - The volume of trace data can be controlled
  - The cost or overhead to collect, transfer and store will be reduced significantly.
  - Helping to solve the scalability issue

- **Provide flexibility**
  - Help user focus on interesting points
  - Can be used as a basis towards automatic performance tuning.
Existing MPI profiling/tracing tool

Program execution

MPI function calls

Trace Buffer

MPI Profiling tool

Derived Metrics

Trace

MPI library
Implementation Example – Programmable MPI profiling/tracing tool

- Program execution
- MPI function calls
- MPI_Finalize
- Record trace?
- Yes
- Trace Buffer
- User defined output
- Derived Metrics
- Output trace?
- Yes
- Trace

MPI Profiling tool

Utility functions

MPI library
Configuration Functions

- **MT_trace_event()**
  - Called by every profiled MPI functions
  - Decide whether the information should be logged

- **MT_output_trace()**
  - Called in MPI_Finalize()
  - Decide whether the node should output the trace collected

- **MT_output_text()**
  - Called in MPI_Finalize()
  - Used for customize performance data output (e.g., user-defined metrics)
Utility functions

- **Software specific info.**
  - Code segment

- **“Tool factor”**
  - Memory usage

- **User preference**
  - User-defined metrics

- **System info.**
  - Node location
Utility Functions

- **Help user configure the profiling/tracing tool**
- **Information include**
  - MPI functions (call counts, size/distance of data transferred)
  - Time
  - Memory usage
  - Compute node environment (coordinates…)
  - Statistics
Utility Functions

- `long long MT_get_mpi_counts(int); /* number of calls for a MPI */`
- `double MT_get_mpi_bytes(int); /* size of data transfer for a MPI */`
- `double MT_get_mpi_time(int); /* time used for a MPI */`
- `double MT_get_avg_hops(void); /* average hops each MPI message travels */`
- `double MT_get_time(void); /* time from the MPI_Init */`
- `double MT_get_elapsed_time(void); /* time between MPI_Finalize and MPI_Init */`
- `char *MT_get_mpi_name(int); /* name for a MPI */`
- `int MT_get_tracebufferinfo(struct …); /* info for trace buffer */`
- `int MT_get_memoryinfo(struct …); /* info for the stack/heap */`
- `int MT_get_calleraddress(void); /* info for the caller */`
- `int MT_get_callerinfo(int, struct …); /* get caller detailed info */`
- `void MT_get_environment(struct …); /* self id info e.g., mpi rank */`
Example Usage

```c
int MT_trace_event( int id ) {
    /* collect call count distribution */
    for ( i=0; i< env.nmpi; i++ )
        event_count[i] = MT_get_mpi_counts(i);

    if ( compare_distribution( event_count, env.nmpi, 0.5 ) == 1 )
        return 0; /* no trace recording if
                   call count distribution stays the same */
    else
        return 1; /* record trace if new call count distribution */
}

int MT_output_trace( int rank ) {
    if ( rank < 32 ) return 1;
    else return 0;
}
```
Modular I/O (MIO)
Modular I/O (MIO)

- **Addresses the need of application-level optimization for I/O.**
- **Analyze and tune I/O at the application level**
  - For example, when an application exhibits the I/O pattern of sequential reading of large files
  - **MIO**
    - Detects the behavior
    - Invokes its asynchronous prefetching module to prefetch user data.
- **Work in progress - Integration into HPC Toolkit with PeekPerf capabilities**
  - Source code traceback
  - Future capability for dynamic I/O instrumentation
Modular I/O Performance Tool (MIO)

- **I/O Analysis**
  - Trace module
  - Summary of File I/O Activity + Binary Events File
  - Low CPU overhead

- **I/O Performance Enhancement Library**
  - Prefetch module (optimizes asynchronous prefetch and write-behind)
  - System Buffer Bypass capability
  - User controlled pages (size and number)

- **Recoverable Error Handling**
  - Recover module (monitors return values and error + reissues failed requests)

- **Remote Data Server**
  - Remote module (simple socket protocol for moving data)
Performance Visualization (work in progress)
MSC.Nastran V2001

Benchmark:
SOL 111, 1.7M DOF, 1578 modes, 146 frequencies, residual flexibility and acoustics. 120 GB of disk space.

Machine:
4-way, 1.3 GHz p655, 32 GB with 16 GB large pages, JFS striped on 16 SCSI disks.

MSC.Nastran:
V2001.0.9 with large pages, dmp=2 parallel=2 mem=700mb
The run with MIO used mio=1000mb

6.8 TB of I/O in 26666 seconds is an average of about 250 MB/sec
ABAQUS Standard v6.3-4

Elapsed Time (seconds)

- 5 M dof, 36 GB fct file
- 11.5 M dof, 80 GB fct file

Engine models
Parallel direct solver
16 POWER4 processors

with MIO
w/o MIO
PompProf
“Standard” OpenMP Monitoring API?

• Problem:
  – OpenMP (unlike MPI) does not define standard monitoring interface (at SC06 they accepted a proposal from SUN and others)
  – OpenMP is defined mainly by directives/pragmas

• Solution:
  – **POMP**: OpenMP Monitoring Interface
  – Joint Development
    • Forschungszentrum Jülich
    • University of Oregon
  – Presented at EWOMP’01, LACSI’01 and SC’01
Profiling of OpenMP Applications: POMP

- Portable cross-platform/cross-language API to simplify the design and implementation of OpenMP tools

- POMP was motivated by the MPI profiling interface (PMPI)
  - PMPI allows selective replacement of MPI routines at link time
  - Used by most MPI performance tools (including MPI Profiler/Tracer)
POMP Proposal

- Three groups of events
  - **OpenMP constructs and directives/pragmas**
    - Enter/Exit around each OpenMP construct
      - Begin/End around associated body
    - Special case for parallel loops:
      - ChunkBegin/End, IterBegin/End, or IterEvent instead of Begin/End
    - “Single” events for small constructs like atomic or flush
  - **OpenMP API calls**
    - Enter/Exit events around `omp_set_*_lock()` functions
    - “single” events for all API functions
  - **User functions and regions**
    - Allows application programmers to specify and control amount of instrumentation
Example: POMP Instrumentation

```c
1:   int main() {
2:       int id;
3:       POMP_Init();
4:       
5:       { POMP_handle_t pomp_hdl = 0;
6:               int32 pomp_tid = omp_get_thread_num();
7:               POMP_ParallelEnter(&pomp_hdl, pomp_tid, -1, 1,
8:                       "49*type=prgion*file=demo.c*slices=4,4*elines=8,8**");
9:       
10:       #pragma omp parallel private(id)
11:       {
12:           int32 pomp_tid = omp_get_thread_num();
13:           POMP_ParallelBegin(pomp_hdl, pomp_tid);
14:           id = omp_get_thread_num();
15:           printf("hello from %d\n", id);
16:           POMP_ParallelEnd(pomp_hdl, pomp_tid);
17:       }
18:       POMP_ParallelExit(pomp_hdl, pomp_tid);
19:       }
20:       POMP_Finalize();
21:   }
```
SIGMA-POMP: Performance Monitoring Interface for OpenMP based on PSIGMA Instrumentation

- **Approach**
  - A POMP implementation using pSigma’s binary instrumentation and rewriting
  - Built on top of pSigma
    - Modifies the binary with performance instrumentation
    - No source code or re-compilation required
POMP Profiler (PompProf)

• Profiler for OpenMP application implemented on top of SIGMA-POMP

• Generates a detailed profile describing overheads and time spent by each thread in three key regions of the parallel application:
  – Parallel regions
  – OpenMP loops inside a parallel region
  – User defined functions

• Profile data is presented in the form of an XML file that can be visualized with PeekPerf
Interactive Performance Debugger
Interactive Performance Debugger

- Control instrumentation from the visualization GUI: one complete framework for performance analysis
- Operate on the source code but perform modifications on the binary
- Debugger-like interface
- Automatically display collected data
- Refine instrumentation (iterative tuning)
- Comparison between data and between multiple runs
- Graphics capabilities (tables, charts)
- Query language for “what-if” analysis
Structure of the HPC toolkit

- Binary Application
  - PeekPerf GUI
    - Communication Profiler
    - Memory Profiler
    - CPU Profiler
    - I/O Profiler
  - Visualization
    - Query
    - Analysis
  - pSigma
    - Instrumented Binary
      - Binary instrumentation
      - execution
Peekperf Main Interface
Peekperf Main Interface (Cont.)
Summery

- The IBM HPC Toolkit provides an integrated framework for performance analysis
- Support iterative analysis and automation of the performance tuning process
- The standardized software layers make it easy to plug in new performance analysis tools
- Operates on the binary and yet provide reports in terms of source-level symbols
- Provides multiple layers that the user can exploit (from low-level instrumentations to high-level performance analysis)
- Full source code traceback capability
- Dynamically activate/deactivate data collection and change what information to collect